JOINT RMIT AND CSIRO PARALLEL SYSTEMS ARCHITECTURE PROJECT

DESIGN CONSIDERATIONS FOR A HIGH PERFORMANCE DATAFLOW MULTIPROCESSOR

D. Abramson
G.K. Egan (RMIT)

TR-FA-88-04 (RMIT TR 112073R)
August, 1988

(RMIT refers to the Department of Communication and Electronic Engineering, Royal Melbourne Institute of Technology, Box 2476V, Melbourne VIC 3001 Australia)

Abstract

This report discusses some of the design decisions in the implementation of a high performance dataflow multiprocessor being constructed by CSIRO and RMIT. The multiprocessor consists of a number of processors which directly execute directed dataflow graphs rather than sequential von Neumann programs. The processors are connected via a high speed multistage interconnection network. The target performance for each processor is a sustained diadic node evaluation rate of 5 millions instructions per second (MIPS), sustained monadic rate of 10 MIPS and a sustained diadic vector rate of 10 MIPS.

1. INTRODUCTION

The RMIT/CSIRO Parallel Systems Architecture Project commenced in May 1986. It is a joint collaborative project between the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organisation, Division of Information Technology. The purpose of the Project is to investigate parallel algorithms, methodologies, languages and architectures, and in particular architectures based on the dataflow model of parallel computation model [1]. The variant of dataflow model being studied is that first proposed in 1976 by Egan at Manchester University, UK [2] and subsequently further developed at RMIT. A multiprocessor emulation facility is available for high speed interpretation of the programs as well as a conventional discrete event simulation of the architecture. Compilers for a number of dataflow-like languages are being developed. Work is currently proceeding on the design of processing elements for a high speed multiprocessor.

The purpose of this paper is to discuss some of the technical design trade offs in the construction of the high speed multiprocessor. The target performance for each processor is a sustained diadic node evaluation rate of 5 millions instructions per second (MIPS), sustained monadic rate of 10 MIPS, and a sustained diadic vector rate of 10 MIPS.

2. DATAFLOW MULTIPROCESSORS

Dataflow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of evaluation of the nodes (or instructions) in the graph is determined by the availability of their operands rather than the strict sequencing of instructions in a von Neumann machine. Consequently the program statements are executed in a non-deterministic manner, and concurrency is obtained if more than one node executes at the same time. Figure 1 shows a sample data-flow graph for an arithmetic expression and Figure 2 shows a model for the hardware required to execute such data-flow programs. In this hardware, the program graph is distributed to the processing elements so that the computation of A*B can proceed at the same time as C*D. The results of a computation are sent from the processor that holds the source node to the processor that holds the destination node. When the result arrives at the destination processor it waits in the matching unit until all of the operands for the destination node are ready before the result is computed. Thus the addition is performed when both A*B and C*D have been computed and the division is computed once the addition has completed.

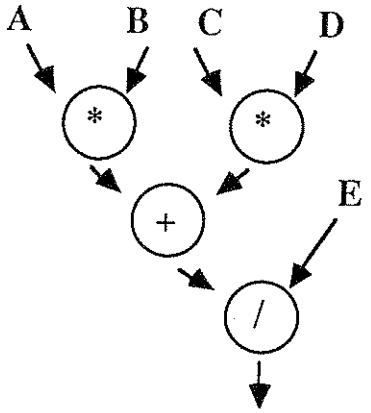
The RMIT dataflow machine differs from other dataflow architectures in the following ways:

- The architecture uses a hybrid static evaluation model with a dynamic model.
- The machine supports vector and list operations
- Graphs are partitioned and the partitions are allocated statically to processing-elements.
- Storage nodes are provided to allow the graph to retain 'semi-permanent' information.
- An Object Store is provided for large structures and persistent objects (e.g.files).

Many of these attributes affect the design of the processors. The effect will be discussed throughout the paper.

3. THE RMIT/CSIRO DATAFLOW ARCHITECTURE

The RMIT hybrid evaluation model has been designed to provide efficient implementation of algorithms which require temporal ordering of data sets, as in many DSP applications, as well as highly concurrent scientific codes. The machine enforces data and temporal dependencies by providing token queuing on the arcs of instructions. Data tokens are consumed from arcs in the order that they are received, thus they move through a graph in the original order that they were introduced. In this case only one instance of the node exists, although the data moves through in a pipelined manner. The machine also allows many concurrent instances of a particular node by tagging the data. In this mode the many tokens on an arc are consumed in an order dictated by the arrival order of partners on the opposite arc. Concurrency is generated because the tokens can be distributed onto different processing elements even though they pertain to the same instruction.



$$F = (A * B + C * D)/E$$

Figure 1 - A Dataflow Graph.

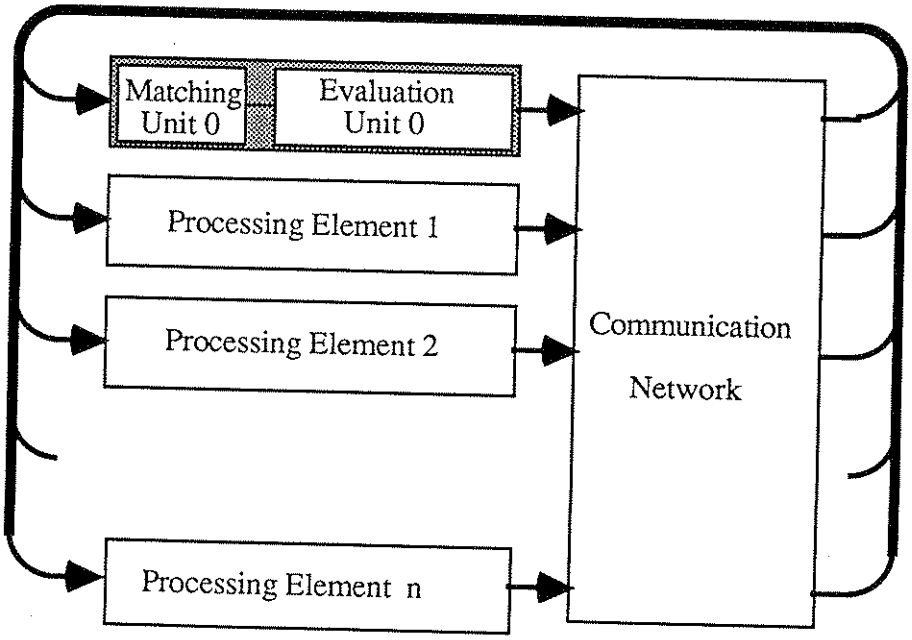


Figure 2 - Hardware Model.

The effect of the hybrid structure on the machine design is that the matching unit must be capable of supporting both tagged token matching as well as token queueing. The advantages are that code which does not require tagging does not need tagging and untagging instructions, and that the matching store performance is better when it is nearly full. These advantages as well as disadvantages are discussed in [4]. The tags required by the dynamic evaluation model are chosen to be very large so that it is not necessary to re-use them. This removes the need for complex tag management hardware, but does require a larger than normal tag field. This has some impact on the design of the matching unit as well as the evaluation unit.

The RMIT machine provides supports operations on vectors as well as scalars. The instruction set uses generic instructions which operate on many different types. For example, if two integers are sent to an add node, then the result is an integer. However, if two floating point number are added the result is a floating point number. In the same way, if two vectors are sent to an add node, then each element of the vectors are added, and the result is a vector. The implementation of vectors as a basic data type allows conventional vector pipelining techniques to be used for improved performance. The machine must be able to match, build and transmit vectors as basic types, which affects both the matching unit design as well as the evaluation unit.

List operations are supported by using attributes of the hybrid model to keep list structures in order. It is possible to operate on a list, construct a list and disassemble a list. Start and end of list tokens delimit lists, including lists of lists. It is possible to mix lists and scalar operations on the same instruction. The implemetation of lists affects the design of the matching unit because they are supported by special matching functions.

Most instructions in the architecture allow an arbitrary number of output destinations. This affects the design of the evaluation unit because it needs to keep a list of destination addresses, and have them ready in time for token dispatch. In most cases a multiple output node is faster and uses less resources than the more common duplicate trees [5].

The storage node is an instruction which retains a single token which may then be re-read on demand. The data is stored in the matching unit for fast access. This has impact on the design of the matching unit because it does not necessarily mean that a token is removed from the matching unit when a match occurs. Large amounts of data are not placed in storage nodes because the matching memory space is a critical resource, and loss of matching space can severly compromise the efficency of the machine. The object storage mechanism allows storage of large or persistent data structures and is implemented by the evaluation unit. Objects can either be direct read write structures with no automatic synchronisation between reads and writes, or can be I-structures [6]. I-Structures provide read before write synchronisation, and prohibit write after write operations, to guard against indeterminate results.

The code partitioning scheme used by the machine distributes work around the multiprocessor giving excellent work load distribution. The effect of this scheme is that there is very little locality as most tokens are required on a different processor from the one which generated them. Later we discuss a network capable of meeting the high bandwidth required for the target evaluation rates.

In the following sections we will discuss the details of the processing elements with emphasis on the processes associated with operand matching.

4. PROCESSOR STRUCTURE

Figure 3 shows the overall structure of a processing element. The element is split into two main functional units. The matching unit accepts tokens as they arrive from the network, and builds work packets for evaluation. Each work packet consists of a function name, a process number, a node number, a colour (or tag), and up to two operands and their types. The evaluation unit applies the function to the operands and sends the results through the network to their destinations.

The matching unit has a large input queue for holding tokens which have arrived from the network, but have not been matched or stored. This queue needs to be sufficiently large to hold excess tokens generated whilst a computation is growing.

Tokens are held in a matching memory whilst they await their partner. This memory behaves as a large associative store, retrieving tokens based on their node numbers (22 bits) and their colours (38 bits). Because the key field is large, a fully associative memory is impractical, and a token cache coupled with a secondary hash table in bulk memory is used.

To account for variation in the matching-class and the evaluation-function times, from token to token and function to function, a small queue of work packets is maintained between the units.

In the evaluation unit simple functions are handled directly in one machine cycle, with more complex operations being supported by microcode. In the normal case the function from the evaluation queue is applied to the ALU's and the operands are processed directly. The results are passed to the dispatch section of the evaluation unit for transmission over the network.

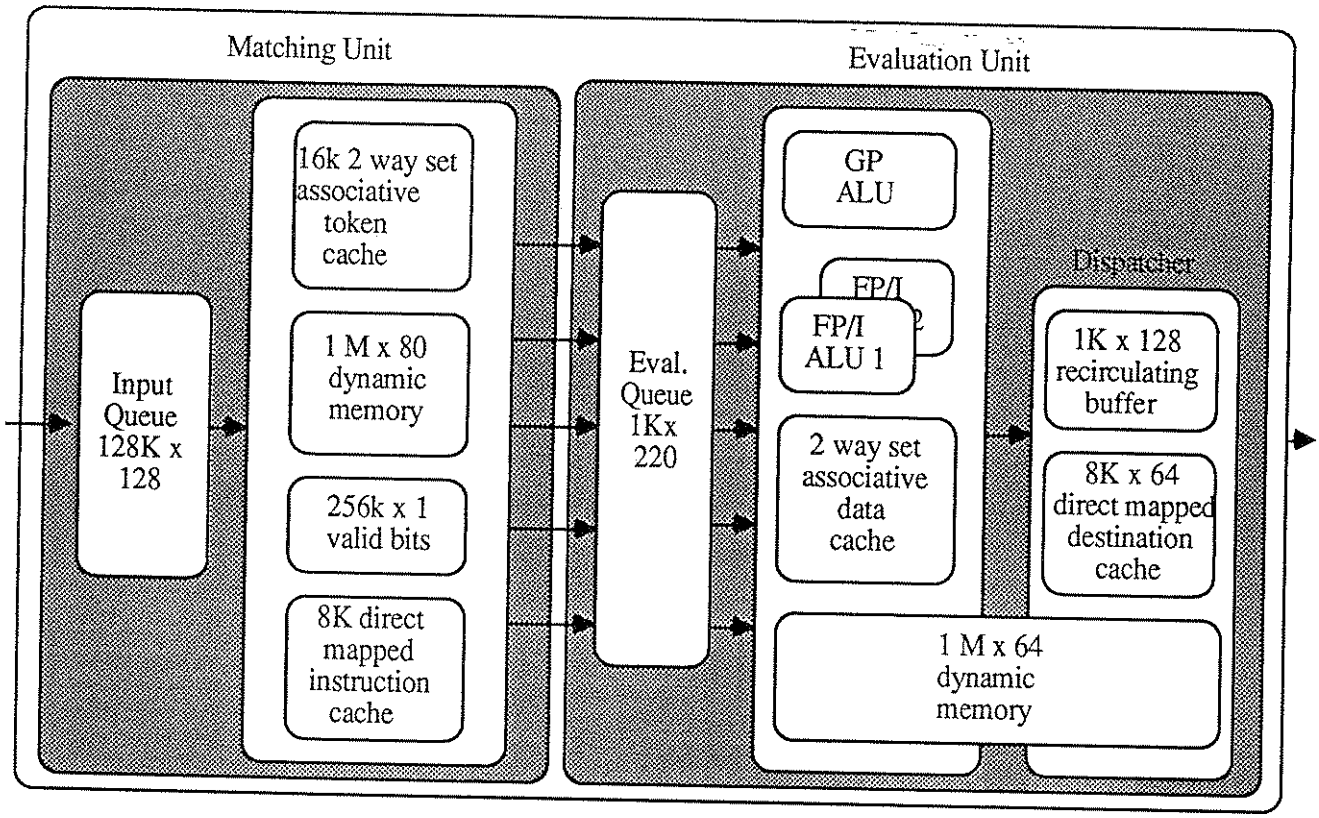


Figure 3 - Processing Element Structure

4.1 Matching Unit

The matching unit is responsible for detecting when both operands of a diadic node have arrived, or when one operand of a monadic node has arrived, and building a work packet for the evaluation unit. The matching unit in the RMIT machine has to perform match operations on both tagged (or coloured) tokens as well as uncoloured tokens. When a tagged token arrives the match unit searches for another token with the same tag and node number. If one is present, and it is for the opposite input point, then it is consumed. If one is present but for the same input point, then the new token is added to the end of a queue. When a token is removed because of a match it is always removed from the head of the queue. When a token arrives without a colour it behaves as though the colour was zero.

The matching unit also handles storage nodes and list operations. In the case of a storage node the data may not be removed from the matching store when a match occurs and can be read out repeatedly until the node is reset or the data overwritten. List operations are qualified by detecting start and end of list tokens.

4.11 Input Queue Design

The input queue poses a design problem because it needs to be both large and fast. It must be capable of meeting the peak transfer rate of the network, but must also be able to hold most of the tokens generated during the buildup phase of a computation. Commercial queue and queue management chips are typically quite small and do not support large data sets.

The input queue has been designed to meet a peak network and processor rate of inserting and removing one token every 50 nSec (for discussion of this rate see Section 4.3). It is constructed using a four way interleaved memory structure from 100 nSec memory devices as shown in Figure 4.

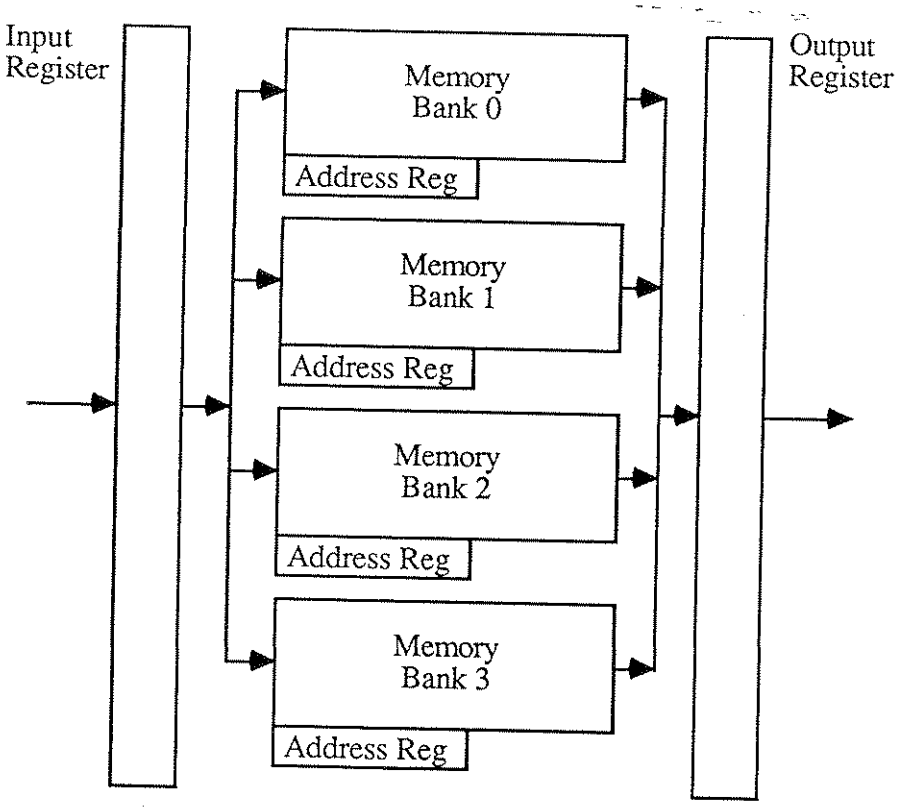


Figure 4 - Interleaved Input Queue

Tokens are composed of one or more 128 bit words. The format of a single word token is shown in Figure 5. The first word of a token contains a monadic match flag, processor number, process number, node number, input point, colour, type and 40 bit data field. The monadic match flag indicates whether a match is required, or whether the token can be passed directly to the evaluation queue; the process number is used to distinguish different users, or tasks, in the machine; the input point indicates which input of the node the token is directed to; the type field indicates the type of the data. The data field is sufficient to hold the most common data types.

If a token consists of multiple words, as in the case of vectors or record structures, then subsequent words hold the data fields. In the case of a vector, each word following the first word would, in the case of 32 bit reals, contain four elements of the vector. Thus, a four element vector only requires 2 words. Records are packed into two or more token words.

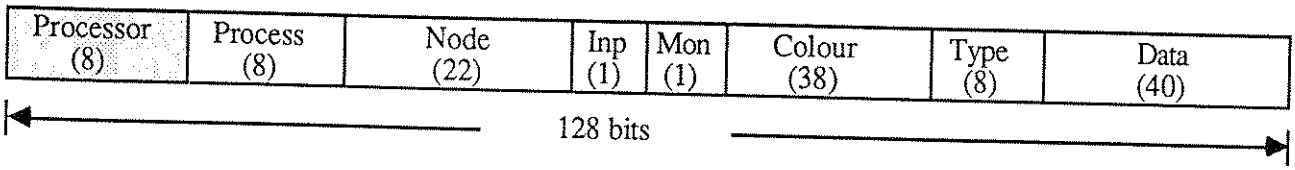


Figure 5 - A Token

4.12 Matching Process

When a token arrives a high speed associative search must be made to determine whether a partner is available. The RMIT machine holds waiting tokens in a hash table. The node and colour fields are hashed into a primary table, as shown in Figure 6. All synonyms are stored in a linked list. When a token arrives the entire chain is searched, and if not found, it is placed at the head of the synonym chain. If a queue is not present, then the entry holds the data directly. If a queue is present, then it contains head and tail pointers to the queue.

The main hash table is designed to be sparsely occupied, so that the probability of an overflow chain is very small. In the RMIT machine, this table is large enough to hold 4 times the maximum number of tokens, thus the chance of an overflow chain is 0.125 [7].

When a tag is not present, the hash table becomes a direct access table, indexed by node number. In this case the overflow chain would only contain one entry, although it is still necessary to verify the node number in the case where a graph contains both coloured and uncoloured tokens. The hashing function combines the node number with the colour field using a multi-way exclusive-or function. When the colour is not present, only the node number is used.

In order to speed access to this matching table, a special token cache holds the most recently written tokens. Thus the hash table is only accessed when the token cache misses.

4.13 Token Lifetimes and Caching

Conventional processors make use of locality to speed up memory accesses by storing the most recently used data items in a high speed cache memory. Such caches work because once a location has been accessed it is a likely candidate for future references. Typical cache hit rates can be very high. However, the matching process in a dataflow machine states that a token either matches with its partner, or is stored and awaits a partner. Once retrieved, a token is normally discarded from the match unit. Thus, the locality exhibited in conventional machines is not present in dataflow machines.

Recent research has indicated that there is actually a significant amount of temporal locality in the arrival times of tokens in a dataflow machine. Once a token is stored in the match unit, its partner will usually arrive shortly after the original token. Thus, it is possible to place tokens in high speed memory when they arrive in the expectation that their partner will arrive within a short time interval; this approach was first proposed by Brobst in [3]. When the token cache becomes full older tokens are retired to some other main memory based associative store. A difficulty with this scheme is that only half of the incoming tokens will match, thus only half of the searches can be performed in the fast cache. In the other cases the secondary memory must still be searched to determine whether the token's partner is already present. Brobst proposed a technique where tokens are always placed in the cache if their partner is not in the cache even though it may have retired to main memory. However, when they are ejected from the cache into main memory a search could take place. This scheme allows matching tokens to remain unmatched even though both are present, and could lead to difficult situations including processor starvation and deadlock.

The RMIT machine uses a cache to hold recently unmatched tokens, but also uses a set of valid bits in high speed memory to indicate whether the partner may be in main memory. The valid bits are taken from the main hash table structure, and indicate whether there are any tokens in the particular overflow chain of the main hash table. If the valid bit is set then the token's partner may be present, and a search is required before the token can be inserted in the cache. By making the hash table large enough, the probability of false triggers can be made acceptably low.

4.14 Instruction Cache

The evaluation patterns of dataflow programs are similar to conventional von Neumann programs, and program loops exhibit significant temporal and spatial locality. The RMIT machine uses a separate direct mapped instruction cache for holding the most recently used instructions. It contains the function code as well as any literal operands attached to the instruction. The cache is read only, and is loaded from main memory when an instruction is not found. If a literal is present then it is loaded into the cache at the same time as the instruction.

4.15 Matching of Queued, Record and Vector Tokens

The token cache is implemented as a two way set associative cache memory, with a fixed size data field of 48 bits. This will allow 32 bits data and 40 bit addresses to be stored directly in the cache in the case when there is no queue present on an input. When a queue is formed the entry is used to hold head and tail pointers to the queue (24 bits each), which is then placed in secondary

memory. In most cases a queue will not be present even in static code, so the cache can process operands at the peak rate of 10 million match/mismatches per second.

The architecture also supports records and vectors as basic tokens. These tokens are of varying sizes, and thus cannot be stored directly in the token cache. When a record or vector token is received an entry for the token is written in the cache, but the data area holds a pointer to the token contents, which are then stored directly in main memory. Each word of the structure is stored in contiguous space, and can be read out sequentially, faster than normal random accesses to the memory, by using page mode dynamic memories. Thus, even though the main memory is slower than the cache memory, the information can be saved and retrieved quickly. In the case of vectors, once the startup cost has been incurred, it will be possible to retrieve 2 elements of the vector every 100 nSec, thus giving a peak vector transfer rate of 20 million elements per second. This will then yield a sustained diadic vector evaluation rate of 10 MIPS.

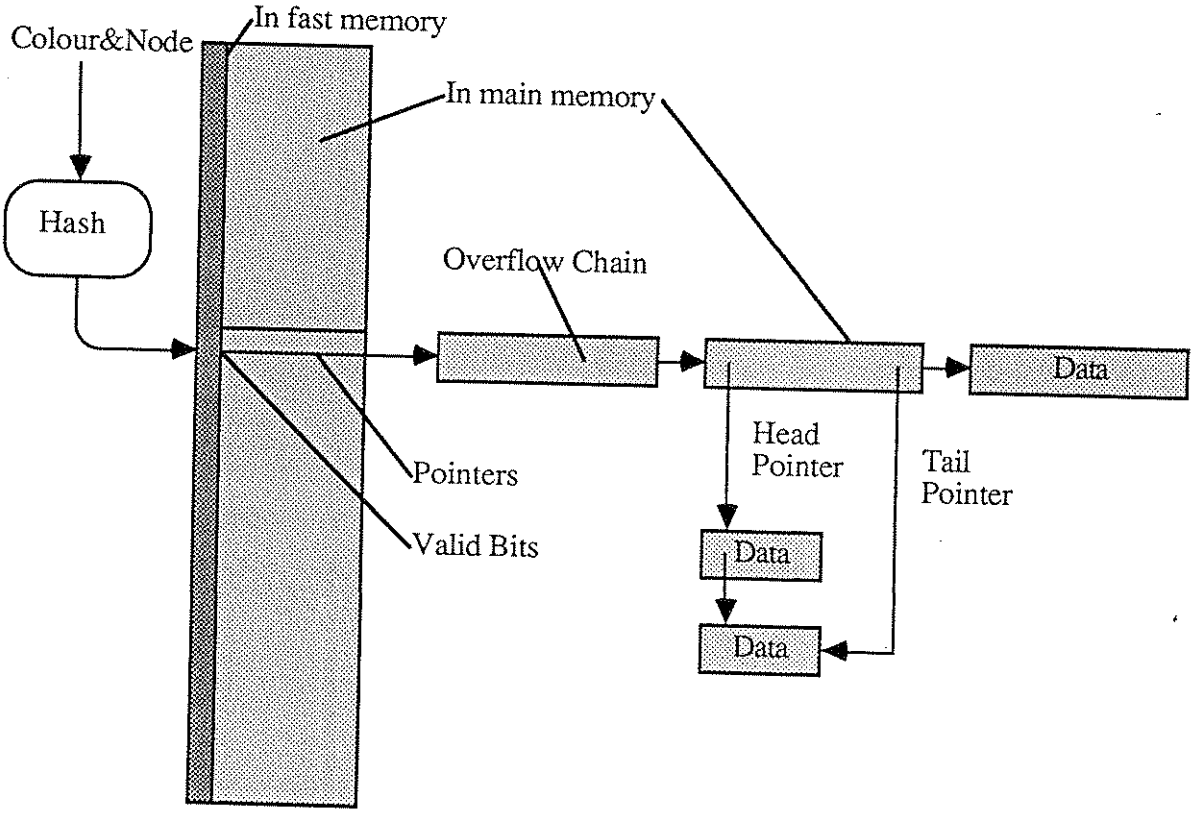


Figure 6 - Hash Table Structure

4.16 Controlling the Match Process

There is a significant difference in the control sequences required when a token is placed in the cache, and when a token must be inserted in main memory. The matching unit control system is optimised so that the cache control is performed by a high speed state machine, but the more complex operations involving main memory are controlled by a microprogrammed control unit. The microprogrammed control unit normally spins in a microcode loop, issuing an instruction which allows the cache control unit to match or mismatch simple tokens. However, when an exception occurs the microprogrammed control unit exits the loop and handles the situation directly.

This technique has the advantage that the most common match operations which can be handled in one machine cycle, execute at the maximum rate, and the more complex and less common cases can tbe dealt with by more general, albeit slower, microcode.

4.2 Evaluation Unit

The evaluation unit supports computational, organisational and structure accessing functions. The major elements of the unit are discussed in the following sections.

4.21 Evaluation Queue

A queue is placed between the matching unit and the evaluation unit to compensate for bursts of monadic or diadic matches and for data dependant variations in operation times of the matching unit and the evaluation unit. If a token is destined for a monadic node, then the matching unit can forward work packets at the rate of 10 million per second. However for normal diadic nodes the rate falls to 5 million per second or less for more complex match-classes and complex data types. Lists, vector and record matches have greater match times due to their size and the reliance on the main memory rather than the cache. The operation times of the evaluation unit vary depending on the complexity of the function to be performed, the datatypes of the operands, whether a stored object access is required, and the number of result destinations.

Each entry in the evaluation queue is a complete work packet. It contains the function name, process number, nodenumber, colour, operand types and operand data. An entry is shown below in Figure 7. The queue is small relative to the input queue, and will be constructed from commercially available 1K deep queue devices.

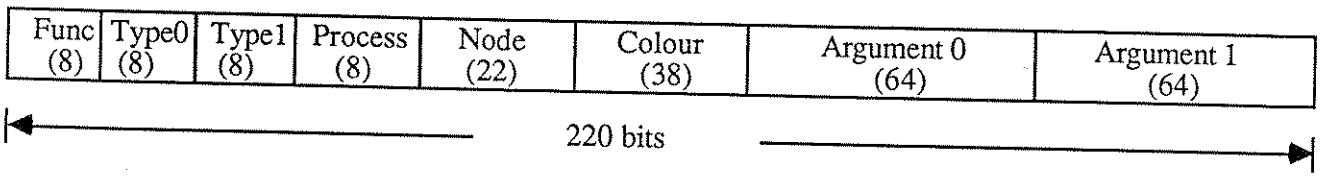


Figure 7 - Evaluation Queue Entry

4.23 Function Evaluation

Function evaluation will be supported by a general purpose ALU and up to two floating point/ integer/logical ALUs. The output of the evaluation queue can be sent to one or more of the ALUs for processing.

The evaluation unit, as for the matching unit will evaluate most functions using a data driven control scheme. The greater richness of functions and operand types increases the difficulty of using this technique however the more frequently occurring functions can be executed in one 200 nSec cycle e.g. multiply two real operands. In more complex functions or where type coercion is required the instruction may require a few to many cycles; in this case a microcode sequence is started to manage function evaluation. The type and function fields are used as an index into the microcode so that the correct sequence can be started without an expensive sequential decoding process.

4.22 Token Dispatch

Nodes can have multiple outputs, thus each node description has a list of destination addresses. A direct mapped cache is used to hold pairs of destinations. If more than two destinations are required then the second destination is used as an indirection pointer to the main memory where the balance of the destinations for the node are stored contiguously. In the normal case two destinations are retrieved while a function is being evaluated.When a token is ready for dispatch, the first destination address is merged with the result data before it is written to the network and a copy of the result token is simultaneously written to a recirculating buffer. If there are two destinations then the second destination is merged and the result is rewritten 50nS later (subject to network contention). If three or more destinations are required then access to main memory is commenced in parallel with the first token being written. Subsequent destination pairs are then returned every 200 nSec. permitting a 100 nSec rate to be maintained for three or more destinations. The dispatcher operates concurrently with the ALU's.

4.23 Vector Function Evaluation

Vectors are passed to the evaluation unit as two 64 bit slices of the original token word, each slice may contain more than one datum e.g. two 32 bit reals. With two floating point processors the evaluation unit can operate on the two vector elements in each operand concurrently. The sustained vector rate will therefore be 10 MFLOPS, compared with the 5 MIPS scalar rate.

4.24 Object Store

The excution unit is responsible for maintaining the object store. It can create, destroy and access objects. An object may reside either entirely within one processing element, or can be distributed across the multiprocessor. The latter form which would most commonly be used for vector and array structures, reduces structure contention but increases latency. A more detailed description of object store operations will be the subject of a later document.

4.3 Communication Network

The dataflow multiprocessor is composed of a number of identical processing elements connected by a high speed multistage network. Workload is distributed by two randomisation processes. One allocates the instructions statically at compile time, and the other distributes the token traffic randomly at run time. The combination of the two yields excellent work load distribution [5]. However, the disadvantage is that all tokens generated must be transmitted over the network. A major feature of dataflow machines is that they can tolerate very high levels of latency in the network because a processor does not remain blocked whilst it waits for data [8]. However, the tokens must still arrive at a rate which allows the processor to achieve its peak instruction rate. For example, a 5 MIPS processor must receive 2 tokens for diadic nodes every 200 nSec, or one token every 100 nSec. If the nodes are monadic, then only half this rate is required. Thus, the network has to have a very high bandwidth, but may be many stages deep and thus have a significant latency without unduly effecting performance.

4.31 Switch Design

The RMIT machine will use a buffered synchronous multistage network built from 4 by 4 cross bar switches. The switches operate on a basic cycle time of one transfer every 50 nSec, which is twice the rate actually required to maintain the processor performance. The reason for this is that in a multistage stage network, the chance of a token moving through the network without contention diminishes as the number of stages increases. The probability of acceptance, falls to about 0.5 for a 3 level network of 4 by 4 switches. Thus, to achieve a transfer every 100 nSec, the switches must actually operate at 50 nSec intervals. The same rate applies to the input and output queues. A 3 level network would support 64 processors providing a sustained vector performance of 640 MFLOPS.

The prototype network will be constructed from conventional PAL devices, and is expected to achieve the target speed easily. Later versions of the switches will be implemented either from gate array technology, or commercially available switches should they be available. An important observation is that the network can be constructed from the same speed devices as those used in the processing elements. In this way, the entire system can be scaled as faster logic becomes available.

5. CONCLUSION

In this paper we have discussed some of the design issues considered during the construction of a high speed dataflow multiprocessor. We have illustrated the sections which have the most effect on performance, and have indicated that it is possible to construct these from current conservative technology. With faster logic families and/or semi-custom silicon implementations of sections of the processor, a much faster processing element could be constructed.

It is anticipated that a prototype processing element will be operating early in 1989. Subsequent work will be to develop a small multistage network, and the construction of a 4 processor multiprocessor. Future work will be involved in measurement of system performance, and considerations involved in mapping sections of the processors onto silicon.

Acknowledgements

The authors wish to acknowledge the support of the project team, in particular M. Rawling, N. Webb, P. Whiting and A. Young. Special thanks go to Mark Rawling and Neil Webb who reviewed earlier drafts of this paper. The Parallel Systems Architecture Project at RMIT is being supported by the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) under an Information Technology joint research project.

References

[1] D.Abramson and G.K. Egan, " An Overview of the RMIT/CSIRO Parallel Systems Architecture Project", TR 112065R, Department of Communication and Electronic Engineering", Royal Melbourne Institute of Technology, Aug. 1987.
Proceedings of 1988 Australian Computer Science Conference, Brisbane, Feb 1988.
Republished in Australian Computer Journal, August 1988.

[2] G.K. Egan, "Data-flow: Its Application to Decentralised Control", Ph.D. Thesis, Department of Computer Science, University of Manchester, 1979.

[3] S. Brobst, "Instruction Scheduling and Token Storage Requirments in a Dataflow Supercomputer", CSG-Memo-264, MIT, May, 1986.

[4] D. Abramson and G.K. Egan "The RMIT Data Flow Computer: A Hybrid Architecture", Royal Melbourne Institute of Technology Technical Report, TR-112-057R, 1987. To appear in The Computer Journal.

[5] M. Rawling, "Implementation and Analysis of a Hybrid Dataflow System, M.Eng. Thesis, Royal Melbourne Institute of Technology, Dec.1987.

[6] Arvind and R.E. Thomas, "I-Structures: An Efficient Data Structure for Functional Languages", MIT/LCS/TM-178, Laboratory for Computer Science, MIT, Sept. 1981.

[7] Morris, R. "Scatter Storage Techniques", Comm. ACM, Jan 1968, pp 38-43.

[8] Arvind and R.A. Iannucci. "Two Fundamental Issues in Multiprocessing", Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Enginerring, Bonn-Bad Godesberg, W Germany, June 25-29, 1987.